HIERARCHICAL PROGRAMMATIC REINFORCEMENT LEARNING VIA LEARNING TO COMPOSE PROGRAMS

Guan-Ting Liu[1*] **& En-Pei Hu**[2*] **& Pu-Jen Cheng**[1] **& Hung-yi Lee**[2] **& Shao-Hua Sun**[2]
[1]Department of Computer Science and Information Engineering
[2]Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
{f07944014, r11921042, pjcheng, hungyilee, shaohuas}@ntu.edu.tw

ABSTRACT

Aiming to produce reinforcement learning (RL) policies that are human-interpretable and can generalize better to novel scenarios, Trivedi et al. (2021) present a method (LEAPS) that first learns a program embedding space to continuously parameterize diverse programs from a pre-generated program dataset, and then searches for a task-solving program in the learned program embedding space when given a task. Despite encouraging results, the program policies that LEAPS can produce are limited by the distribution of the program dataset. Furthermore, during searching, LEAPS evaluates each candidate program solely based on its return, failing to precisely reward correct parts of programs and penalize incorrect parts. To address these issues, we propose to learn a meta-policy that composes a series of programs sampled from the learned program embedding space. By composing programs, our proposed method can produce program policies that describe out-of-distributionally complex behaviors and directly assign credits to programs that induce desired behaviors. We design and conduct extensive experiments in the Karel domain. The experimental results show that our proposed framework outperforms baselines. The ablation studies confirm the limitations of LEAPS and justify our design choices.

1 INTRODUCTION

Deep reinforcement learning (DRL) leverages the recent advancement in deep learning by reformulating the reinforcement learning problem as learning policies or value functions parameterized by deep neural networks. DRL has achieved tremendous success in various domains, including controlling robots Gu et al. (2017); Ibarz et al. (2021); Lee et al. (2019; 2021), playing board games Silver et al. (2016; 2017), and strategy games Vinyals et al. (2019); Wurman et al. (2022). Yet, the black-box nature of neural network-based policies makes it difficult for the DRL-based systems to be interpreted and therefore trusted by human users Lipton (2016); Puiutta & Veith (2020). Moreover, policies learned by DRL methods tend to overfit and often fail to generalize Zhang et al. (2018); Cobbe et al. (2019); Sun et al. (2020); Liu et al. (2022).

To address the abovementioned issues of DRL, programmatic RL methods Bastani et al. (2018); Inala et al. (2020); Landajuela et al. (2021); Verma et al. (2018) explore various structured representations of policies. In particular, Trivedi et al. (2021) present a framework, **L**earning **E**mbeddings for l**A**tent **P**rogram **S**ynthesis (LEAPS), that is designed to produce more interpretable and generalizable policies. Specifically, it aims to produce program policies structured in a given domain-specific language (DSL), which can be executed to yield desired behaviors. To this end, LEAPS first learns a program embedding space to continuously parameterize diverse programs from a pre-generated program dataset, and then searches for a task-solving program in the learned program embedding space when given a task described by a Markov decision process (MDP). The program policies produced by LEAPS are not only human-readable but also achieve competitive performance and demonstrate superior generalization ability.

[*]These authors contributed equally to this work

Despite its encouraging results, LEAPS has two fundamental limitations. *Limited program distribution*: the program policies that LEAPS can produce are limited by the distribution of the pre-generated program dataset used for learning the program embedding space. This is because LEAPS is designed to search for a task-solving program from the learned embedding space, which inherently assumes that such a program is within the distribution of the program dataset. Such design makes it difficult for LEAPS to synthesize programs that are out-of-distributionally long or complex. *Poor credit assignment*: during the search for the task-solving program embedding, LEAPS evaluates each candidate program solely based on the cumulative discounted return of the program execution trace. Such a design fails to accurately attribute rewards obtained during the execution trajectories to corresponding parts in synthesized programs or penalize program parts that induce incorrect behaviors.

This work aims to address the issues of limited program distribution and poor credit assignment. To this end, we propose a hierarchical programmatic reinforcement learning (HPRL) framework. Instead of searching for a program from a learned program embedding space, we propose to learn a meta-policy, whose action space is the learned program embedding space, to produce a series of programs (*i.e.*, predict a sequence of actions) to yield a composed task-solving program. By re-formulating synthesizing a program as predicting a sequence of programs, HPRL can produce out-of-distributionally long or complex programs. Furthermore, rewards obtained from the environment by executing each program from the composed program can be accurately attributed to the program, allowing for more efficient learning.

To evaluate our proposed method, we adopt the Karel domain Pattis (1981), which features an agent that can navigate a grid world and interact with objects. Our method outperforms all the baselines by large margins on a problem set proposed by Trivedi et al. (2021). To investigate the limitation of our method, we design a more challenging problem set on which our method consistently achieves better performance compared to LEAPS. Moreover, we inspect LEAPS' issues of limited program distribution and poor credit assignment with two experiments and demonstrate that our proposed method addresses these issues. We present a series of ablation studies to justify our design choices, including the reinforcement learning algorithms used to learn the meta-policy and the dimensionality of the program embedding space. A detailed discussion of related work can be found in Section A.

2 Problem Formulation

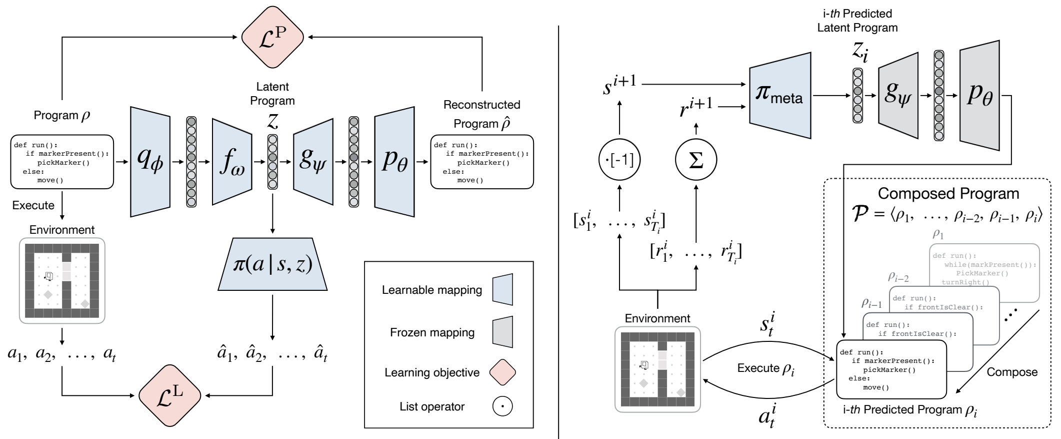
Our goal is to develop a method that can synthesize a domain-specific, task-solving program which can be executed to interact with an environment and maximize a discounted return defined by a Markov Decision Process.

$$
\begin{aligned}
\text{Program } \rho &:= \text{DEF run m(} s \text{ m)} \\
\text{Repetition } n &:= \text{Number of repetitions} \\
\text{Perception } h &:= \text{frontIsClear} \mid \text{leftIsClear} \mid \text{rightIsClear} \mid \\
&\quad\ \text{markerPresent} \mid \text{noMarkerPresent} \\
\text{Condition } b &:= \text{perception h} \mid \text{not perception h} \\
\text{Action } a &:= \text{move} \mid \text{turnLeft} \mid \text{turnRight} \mid \\
&\quad\ \text{putMarker} \mid \text{pickMarker} \\
\text{Statement } s &:= \text{while c(} b \text{ c) w(} s \text{ w)} \mid s_1 ; s_2 \mid a \mid \\
&\quad\ \text{repeat R=}n \text{ r(} s \text{ r)} \mid \text{if c(} b \text{ c) i(} s \text{ i)} \mid \\
&\quad\ \text{ifelse c(} b \text{ c) i(} s_1 \text{ i) else e(} s_2 \text{ e)}
\end{aligned}
$$

Figure 1: The domain-specific language (DSL) for the Karel domain, features an agent that can navigate through a grid world and interact with objects.

Domain Specific Language. In this work, we adapt the domain specific language (DSL) for the Karel domain used in Bunel et al. (2018); Chen et al. (2019); Trivedi et al. (2021), shown in Figure 1. This DSL is designed to describe the behaviors of the Karel agent, consisting of control flows, agent's perceptions, and agent's actions. Control flows such as `if`, `else`, and `while` are allowed for describing diverging or repetitive behaviors. Furthermore, Boolean and logical operators such as `and`, `or`, and `not` can be included to express more sophisticated conditions. Perceptions such as `frontIsClear` and `markerPresent` are defined based on situations in an environment which can be perceived by an agent. On the other hand, actions such as `move`, `turnRight`, and `putMarker`, describe the primitive behaviors that an agent can perform in an environment. A program policy considered in our work is structured in this DSL and can be executed to produce actions based on perceptions.

Markov Decision Process (MDP). The tasks considered in this work are defined by finite-horizon discounted MDPs. The performance of a policy with its rollout (a sequence of states and actions $\{(s_0, a_0), ..., (s_t, a_t)\}$) is evaluated based on a discounted return $\sum_{t=0}^{T} \gamma^t r_t$, where $r_t = \mathcal{R}(s_t, a_t)$ indicates the reward function and T is the horizon of the episode. We aim to develop a method that can produce a program representing a policy that can be executed to maximize the discounted return, *i.e.*, $\max_\rho \mathbb{E}_{a \sim \text{EXEC}(\rho)}[\sum_{t=0}^{T} \gamma^t r_t]$, where EXEC$(\cdot)$ returns the actions induced by executing the program policy ρ in the environment. This objective is a special case of the standard RL objective where a policy is represented as a program and its rollout is obtained by executing the program.



```
def run():
  if markerPresent():
    pickMarker()
  else:
    move()
```

```
def run():
  if markerPresent():
    pickMarker()
  else:
    move()
```

```
def run():
  while(markPresent()):
    PickMarker()
  turnRight()
```

```
def run():
  if frontIsClear():
```

```
def run():
  if frontIsClear():
```

```
def run():
  if markerPresent():
    pickMarker()
  else:
    move()
```

(a) Learning a Program Embedding Space

(b) Learning a Meta-Policy to Compose Programs

Figure 2: **Hierarchical Programmatic Reinforcement Learning. (a) Learning a Program Embedding Space**: a continuously parameterized latent program space can be learned using the program encoder q_ϕ, decoder p_θ, and a neural executor policy π by optimizing the two reconstruction objectives: $\mathcal{L}^P$ and $\mathcal{L}^L$. To reduce the dimensionality of the program embedding space for facilitate task learning, we employ a compression encoder f_ω and a compression decoder g_ψ. **(b) Learning a Meta-Policy to Compose Programs**: given a task described by an MDP, we propose to train a meta-policy π_{meta} to compose a sequence of programs, and yield a task-solving program. Specifically, at each macro time step i, the meta-policy π_{meta} predicts a latent program embedding z_i, which can be decoded to the corresponding program $\rho_i = p_\theta(g_\psi(z_i))$. We then execute the program ρ_i in the environment, which returns the cumulative reward r^{i+1} and the next state s^{i+1} to the meta policy. The meta-policy can synthesize next program ρ_{i+1} based on s^{i+1} to synthesize the task-solving program $\mathcal{P} = \langle \rho_1, .., \rho_{i-2}, \rho_{i-1}, \rho_i \rangle$, until termination.

3 APPROACH

Our goal is to design a framework that can synthesize task-solving programs based on the rewards obtained from MDPs. We adapt the idea of learning a program embedding space to continuously parameterized a diverse set of programs proposed in LEAPS Trivedi et al. (2021). Then, instead of searching for a task-solving program in the learned program embedding space, our key insight is to learn a meta-policy that can hierarchically compose programs to form a more expressive task-solving program. Our proposed framework, dubbed Hierarchical Programmatic Reinforcement Learning (HPRL), is capable of producing out-of-distributionally long and complex programs. Moreover, HPRL can make delicate adjustments to synthesized programs according to rewards obtained from the environment.

Section 3.1 presents how LEAPS learns a program embedding space to continuously parameterize a set of randomly generated programs and describes our proposed procedure to produce a dataset containing more diverse programs. Then, to reduce the dimension of the learned program embedding for more efficient meta-policy learning, Section 3.2 introduces how we compress the embedding space. Finally, in Section 3.3, we describe our method for learning a meta-policy, whose action space is the learned program embedding space, to hierarchically compose programs and yield a task-solving program. An overview of our proposed framework is illustrated in Figure 2 and the algorithm is detailed in Algorithm 1.

3.1 LEARNING A PROGRAM EMBEDDING SPACE

We aim to learn a program embedding space that continuously parameterizes a diverse set of programs. Moreover, a desired program embedding space should be behaviorally smooth, *i.e.*, programs that induce similar execution traces should be embedded closely to each other and programs with diverging behaviors should be far from each other in the embedding space.

To this end, we adapt the technique proposed in LEAPS (Trivedi et al., 2021), which trains an encoder-decoder neural network architecture on a pre-generated program dataset. Specifically, a recurrent neural network program encoder q_ϕ learns to encode a program ρ (*i.e.*, sequences of program tokens) into a program embedding space, yielding a program embedding v; a recurrent neural network program decoder p_θ learns to decode a program embedding v to produce reconstructed programs $\hat{\rho}$. The program encoder and the program decoder are trained to optimize the β-VAE Higgins et al. (2016) objective: $\mathcal{L}^{\mathrm{P}}_{\theta,\phi}(\rho) = -\mathbb{E}_{v\sim q_\phi(v|\rho)}[\log p_\theta(\rho|v)] + \beta D_{\mathrm{KL}}(q_\phi(v|\rho)\|p_\theta(v))$, where β balances the reconstruction loss and the representation capacity of the embedding space (*i.e.*, the latent bottleneck).

To encourage behavioral smoothness, Trivedi et al. (2021) propose two additional objectives. The *program behavior reconstruction loss* minimizes the difference between the execution traces of the given program EXEC(ρ) and the execution traces of the reconstructed program EXEC($\hat{\rho}$). On the other hand, the *latent behavior reconstruction loss* brings closer the execution traces of the given program EXEC(ρ) and the execution traces produced by feeding the program embedding v to a learned neural program executor $\pi(a|v,s)$: $\mathcal{L}^{\mathrm{L}}_{\pi}(\rho,\pi) = -\mathbb{E}[\sum_{t=1}^{H}\sum_{i=1}^{|\mathcal{A}|}\mathbb{1}\{\mathrm{EXEC}_i(\hat{\rho}) == \mathrm{EXEC}_i(\rho)\}\log\pi(a_i|v,s_t)]$, where H denotes the horizon of EXEC(ρ) and $|\mathcal{A}|$ denotes the cardinality of the action space.

We empirically found that optimizing the *program behavior reconstruction loss* does not yield a significant performance gain. Yet, due to the non-differentiability nature of program execution, optimizing this loss via REINFORCE (Williams, 1992) is unstable. Moreover, performing on-the-fly program execution during training significantly slows down the learning process. Therefore, we exclude the *program behavior reconstruction loss*, yielding our final objective for learning a program embedding space as a combination of the β-VAE objective $\mathcal{L}^{\mathrm{P}}_{\theta,\phi}$ and the *latent behavior reconstruction loss* $\mathcal{L}^{\mathrm{L}}_{\pi}$: $\min_{\theta,\phi,\pi}\mathcal{L}^{\mathrm{P}}_{\theta,\phi}(\rho) + \lambda\mathcal{L}^{\mathrm{L}}_{\pi}(\rho,\pi)$, where λ determines the relative importance of these losses.

3.2 Compressing the Learned Program Embedding Space

The previous section describes a method for constructing a program embedding space that continuously parameterizes programs. Next, given a task defined by an MDP, we aim to learn a meta-policy that predicts a sequence of program embeddings as actions to compose a task-solving program. Hence, a low-dimensional program embedding space (*i.e.*, a smaller action space) is ideal for efficiently learning such a meta-policy. Yet, to embed a large number of programs with diverse behaviors, a learned program embedding space needs to be extremely high-dimensional.

Therefore, our goal is to bridge the gap between a high-dimensional program embedding space with sufficient representation capacity and a desired low-dimensional action space for learning a meta-policy. To this end, we propose to learn to compress the program embedding space with an encoder-decoder architecture. Specifically, we employ a compression encoder f_ω that takes the output of the program encoder q_ϕ as input and compresses it into a lower-dimensional program embedding z; also, we employ a compression decoder g_ψ that takes a program embedding as input and decompresses it to produce a reconstructed higher-dimensional program embedding $\hat{v}$, which is then fed to the program decoder p_θ to produce a reconstructed program $\hat{\rho}$.

With this modification, the β-VAE objective and the *latent behavior reconstruction loss* can be rewritten as: $\mathcal{L}^{\mathrm{P}}_{\theta,\phi,\omega,\psi}(\rho) = -\mathbb{E}_{z\sim f_\omega(z|q_\phi(\rho))}[\log p_\theta(\rho|(g_\psi(z)))] + \beta D_{\mathrm{KL}}(f_\omega(q_\phi(z|\rho))\|p_\theta(g_\psi(z)))$, and $\mathcal{L}^{\mathrm{L}}_{\pi}(\rho,\pi) = -\mathbb{E}[\sum_{t=1}^{H}\sum_{i=1}^{|\mathcal{A}|}\mathbb{1}\{\mathrm{EXEC}_i(\hat{\rho}) == \mathrm{EXEC}_i(\rho)\}\log\pi(a_i|z,s_t)]$. We train the program encoder q_ϕ, the compression encoder f_ω, the compression decoder g_ψ, the program decoder p_θ, and the neural execution policy π in an end-to-end manner. We discuss how the dimension of the program embedding space affects the quality of program reconstruction and the performance of synthesized programs in Section 4.4.2.

3.3 Learning a Meta-Policy to Compose the Task-Solving Program

Once an expressive, smooth, yet compact program embedding space is learned, given a task described by an MDP, we propose to train a meta-policy π_{meta} to compose a task-solving program. Specifically, the learned program embedding space is used as a continuous action space for the meta-policy π_{meta}, bounded within the range of [-1.0, 1.0] for each dimension of the program embedding. We formulate the task of composing programs as a finite-horizon MDP whose horizon is $|H|$. At each time step i, the meta-policy π_{meta} takes an input state s^i, and predicts one latent program embedding z_i as action, which can be decoded to its corresponding program ρ_i using the learned compression decoder and program decoder $p_\theta(g_\psi(z_i))$. Then, the program ρ_i is executed with EXEC$(\cdot)$ to interact with the environment for a period from 1 to T^i, yielding the cumulative reward $r^{i+1} = \sum_{t=1}^{T^i} r_t^i$ and the next state $s^{i+1} = [s_1^i, s_2^i, ..., s_{T_i}^i][-1]$ after the program execution. The operator $\cdot[-1]$ returns the last object in the sequence, and we take the last state of the program execution as the next macro input state, i.e., $s^{i+1} = [s_1^i, s_2^i, ..., s_{T_i}^i][-1] = s_{T_i}^i$. Note that the time steps i considered here are macro time steps, and each involves a series of state transitions and returns a sequence of rewards. The environment will return the next state s^{i+1} and cumulative reward r^{i+1} to the agent to predict the next latent program embedding z_{i+1}. The program composing process terminates after $|H|$ steps.

The synthesized task-solving program $\mathcal{P}$ is obtained by sequentially composing the generated program $\langle \rho_i | i = 1...|H| \rangle$, where $\langle \cdot \rangle$ denotes an operator that concatenates programs in order to yield a composed program. Hence, the learning objective of the meta-policy π_{meta} is to maximize the total cumulative return $\mathcal{J}_{\pi_{\text{meta}}}$: $\mathcal{J}_{\pi_{\text{meta}}} = \mathbb{E}_{\mathcal{P}\sim\pi_{\text{META}}}[\sum_{i=1}^{|H|} \gamma^{i-1} \mathbb{E}_{a\sim\text{EXEC}(\rho_i)}[r^{i+1}]$, where γ is the discount factor for macro time steps MDP and a is the primitive action triggered by EXEC(ρ_i).

While this work formulates the program synthesis task as a finite-horizon MDP where a fixed number of programs are composed, we can instead learn a termination function that decides when to finish the program composition process, which is left to future work.

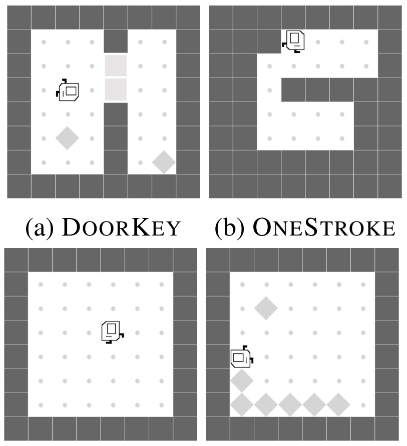
4 EXPERIMENTS

We design and conduct experiments to compare our proposed framework to its variants and baselines.

4.1 KAREL DOMAIN

For the experiments and ablation studies, we adopt the Karel domain (Pattis, 1981), which is widely used in neural program synthesis and programmatic reinforcement learning (Bunel et al., 2018; Shin et al., 2018; Sun et al., 2018; Chen et al., 2019; Trivedi et al., 2021). The Karel agent in a gridworld can navigate and interact with objects (*i.e.*, markers). The action and perception are detailed in Figure 1.

To evaluate the proposed framework and the baselines, we consider two problem sets. First, we use the KAREL problem set proposed in Trivedi et al. (2021), which consists of six tasks. Then, we propose a more challenging set of tasks, KAREL-HARD problem set (shown in Figure 3), which consists of four tasks. In most tasks, initial configurations such as agent and goal locations, wall and marker placements are randomly sampled upon every episode reset.


(a) DOORKEY (b) ONESTROKE

(c) SEEDER (d) SNAKE

Figure 3: **KAREL-HARD Problem Set**: The four tasks require an agent to acquire a set of diverse, goal-oriented, and programmatic behaviors. This is strictly more challenging compared to the KAREL problem set proposed in Trivedi et al. (2021). More details can be found in Section D.

KAREL Problem Set. The KAREL problem set introduced in Trivedi et al. (2021) consists of six tasks: STAIRCLIMBER, FOURCORNER, TOPOFF, MAZE, CLEANHOUSE and HARVESTER. Solving these tasks requires the following ability. *Repetitive Behaviors*: to conduct the same behavior for several times, *i.e.*, placing markers on all corners (FOURCORNER) or move along the wall (STAIRCLIMBER). *Exploration*: to navigate the agent through complex patterns (MAZE) or multiple chambers (CLEANHOUSE). *Complexity*: to perform specific actions, *i.e.*, put markers on the marked grid (TOPOFF) or pick markers on markerd grid (HARVESTER). For further description of the KAREL problem set, please refer to Section D.1.

KAREL-HARD Problem Set. We design a more challenging set of tasks, the KAREL-HARD problem set. The ability required to solve the tasks in this problem set can be categorized as follows: *Two-stage exploration*: to explore the environment under different conditions, *i.e.*, pick up the marker in one chamber to unlock the door, and put the marker in the next chamber (DOORKEY). *Additional*

Table 1: Mean return and standard deviation of all methods across the KAREL problem set over three random seeds. HPRL-PPO outperforms all prior approaches and achieves the maximum score on all tasks. HPRL-SAC completely solves five out of six tasks.

Method	STAIRCLIMBER	FOURCORNER	TOPOFF	MAZE	CLEANHOUSE	HARVESTER
DRL	**1.00** $\pm$ 0.00	0.29 $\pm$ 0.05	0.32 $\pm$ 0.07	**1.00** $\pm$ 0.00	0.00 $\pm$ 0.00	0.90 $\pm$ 0.10
DRL-abs	0.13 $\pm$ 0.29	0.36 $\pm$ 0.44	0.63 $\pm$ 0.23	**1.00** $\pm$ 0.00	0.01 $\pm$ 0.02	0.32 $\pm$ 0.18
VIPER	0.02 $\pm$ 0.02	0.40 $\pm$ 0.42	0.30 $\pm$ 0.06	0.69 $\pm$ 0.05	0.00 $\pm$ 0.00	0.51 $\pm$ 0.07
LEAPS	**1.00** $\pm$ 0.00	0.45 $\pm$ 0.40	0.81 $\pm$ 0.07	**1.00** $\pm$ 0.00	0.18 $\pm$ 0.14	0.45 $\pm$ 0.28
LEAPS-ours	**1.00** $\pm$ 0.00	0.75 $\pm$ 0.43	0.76 $\pm$ 0.10	**1.00** $\pm$ 0.00	0.32 $\pm$ 0.02	0.70 $\pm$ 0.03
HPRL-SAC	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	0.36 $\pm$ 0.13	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00
HPRL-PPO	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00	**1.00** $\pm$ 0.00

Constraints: to perform specific actions under restrictions, *i.e.*, traverse the environment without revisiting the same position (ONESTROKE), place exactly one marker on all grids (SEEDER), and traverse the environment without hitting a growing obstacle (SNAKE). More details about the KAREL-HARD problem set can be found in Section D.8.

4.2 EXPERIMENTAL SETTINGS

Section 4.2.1 introduces the procedure for generating the program dataset used for learning a program embedding pace. The implementation of the proposed framework is described in Section 4.2.2.

4.2.1 KAREL DSL PROGRAM DATASET GENERATION WITH OUR IMPROVED GENERATION PROCEDURE

The Karel program dataset used in this work includes one million programs. All the programs are generated based on the Karel DSL syntax rules with a maximum length of 40 program tokens. While Trivedi et al. (2021) sample randomly to generate program sequences, we propose an improved program generation procedure as follows. We filter out counteracting programs (*e.g.* termination state equals initial state after program execution), repetitive programs (*e.g.* , programs with long common sub-sequences) and programs with canceling action sequences (e.g., `turnLeft` followed by `turnRight`). These rules significantly improve the diversity and expressiveness of the generated programs and induce a more diverse and complex latent program space. More details can be found in Section F.

4.2.2 IMPLEMENTATION

Encoders & Decoders. We use GRU (Cho et al., 2014) layer to implement both the program encoder q_ϕ and the program decoder p_θ mentioned in Section 3.1 with a hidden dimension of 256. The last hidden state of the encoder q_ϕ is taken as the uncompressed program embedding v. This program embedding v can be further compressed to a 64-dimensional program embedding z using the compression encoder f_ω and compression decoder g_ψ constructed by the fully-connected neural network as described in Section 3.2.

Neural Program Executor. The neural program executor π is implemented as a recurrent conditional policy $\pi(\cdot|z, s)$ using GRU layers, which takes the abstract state and the program embedding z at each time step as input and predicts the execution trace.

Meta-Policy. To implement the meta-policy π_{meta}, we use convolutional layers (Fukushima & Miyake, 1982; Krizhevsky et al., 2017) to extract features from the Karel states and then process them with GRU layers for predicting program embeddings. To optimize the meta-policy, we use two popular reinforcement learning algorithms, PPO (Schulman et al., 2017) and SAC (Haarnoja et al., 2018), and report their experimental results as HPRL-PPO and HPRL-SAC, respectively.

More details on hyperparameters, training procedures, and implementation can be found in Section E.

4.2.3 BASELINE APPROACHES

We compare HPRL with the following baselines.

DRL and DRL-abs. Deep RL baselines from Trivedi et al. (2021). DRL observes a raw state (grids) input from the Karel environment, while DRL-abs is a recurrent neural network policy that takes abstracted state vectors from the environment as input. The abstracted state vectors consist of binary values of the current state (*e.g.* [`frontIsClear() == True`, `markerPresent()==False`, `...`]).

VIPER. A programmatic RL method proposed by Bastani et al. (2018). It uses a decision tree to imitate the behavior of a learned DRL policy.

LEAPS. A programmatic RL framework proposed by Trivedi et al. (2021) that uses Cross-Entropy Method (Rubinstein, 1997) to search task-solving program in a learned continuous program embedding space.

Table 2: Mean return and standard deviation of all methods across the KAREL-HARD problem set, evaluated over three random seeds. HPRL-PPO achieves best performance across all tasks.

Method	DOORKEY	ONESTROKE	SEEDER	SNAKE
LEAPS	**0.50** ± 0.00	0.65 ± 0.24	0.51 ± 0.06	0.23 ± 0.10
LEAPS-ours	**0.50** ± 0.00	0.68 ± 0.11	0.56 ± 0.00	0.28 ± 0.08
HPRL-SAC	**0.50** ± 0.00	0.76 ± 0.08	0.32 ± 0.10	0.25 ± 0.06
HPRL-PPO	**0.50** ± 0.00	**0.80** ± 0.03	**0.58** ± 0.10	**0.33** ± 0.12

LEAPS-ours. The LEAPS framework trained on the proposed Karel program dataset described in Section 4.2.1. This is used to compare our proposed program dataset generation procedure with the generation approach used by Trivedi et al. (2021).

More details of these baselines can be found in Section C.

4.3 EXPERIMENTAL RESULTS

We evaluate the performance in terms of the cumulative return of all methods on the KAREL problem set and the KAREL-HARD problem set. The experimental results are presented in Table 1 and Table 2, respectively. The range of the cumulative return is within $[0, 1]$ on tasks without penalty, and within $[-1, 1]$ on tasks with the penalty. Section D describes the detailed definition of the reward function for each task. The performance of DRL, DRL-abs, VIPER, and LEAPS was reproduced with the implementation provided by Trivedi et al. (2021). The average cumulative return and standard deviation of LEAPS-ours, HPRL-PPO and HPRL-SAC on each task are evaluated over three random seeds to ensure statistical significance. The programs synthesized by LEAPS, LEAPS-ours, and HPRL are presented in Section H.

Overall Karel Task Performance. The experimental results on Table 1 show that HPRL-PPO outperforms all other approaches on all tasks. Furthermore, HPRL-PPO can completely solve all the tasks in the KAREL problem set. We also observe that LEAPS-ours outperforms LEAPS on five out of six tasks in the KAREL problem set, showing that the proposed program generation process helps improve the quality of the program embedding space and lead to better program search result.

Overall Karel-Hard Task Performance. To further test the efficacy of the proposed method, we evaluate LEAPS, LEAPS-ours, HPRL-PPO, and HPRL-SAC on the KAREL-HARD problem set. HPRL-PPO outperforms other methods on ONESTROKE, SEEDER, and SNAKE, while all approaches perform similarly on DOORKEY. The complexity of ONESTROKE, SEEDER, and SNAKE makes it difficult for LEAPS to find satisfactory policies from the program embedding space simply by searching. In contrast, HPRL-PPO addresses this by composing a series of programs to increase the expressiveness and perplexity of the synthesized program. We observe that LEAPS-ours achieve better performance than LEAPS, justifying the efficacy of the proposed program generation procedure.

PPO vs. SAC. HPRL-SAC can still deliver competitive performance in comparison with HPRL-PPO. However, we find that HPRL-SAC is more unstable on complex tasks (*e.g.* TOPOFF) and tasks with additional constraints (*e.g.* SEEDER and SNAKE). On the other hand, HPRL-PPO is more stable across all tasks and achieves better performance on both problem sets. Hence, we adopt HPRL-PPO as our main method in the following experiments.

4.4 ADDITIONAL EXPERIMENTS

We design experiments to justify (1) whether LEAPS (Trivedi et al., 2021) and our proposed framework can synthesize out-of-distributional programs (Section 4.4.1), (2) the necessity of the proposed compression encoder and decoder (Section 4.4.2), and (3) the effectiveness of learning from dense rewards made possible by the hierarchical design of our framework (Section B).

4.4.1 SYNTHESIZING OUT-OF-DISTRIBUTIONAL PROGRAMS

Programs that LEAPS can produce are fundamentally limited by the distribution of the program dataset since it searches for programs in the learned embedding space. It is impossible for LEAPS to synthesize programs that are significantly longer than the programs provided in the dataset. This section aims to verify this hypothesis and evaluate the capability of generating out-of-distributional programs. We create a set of target programs of lengths 25, 50, 75, and 100, each consisting of primitive actions (*e.g.* `move`, `turnRight`). Then, we ask LEAPS and HPRL to fit each target program based on how well the program produced by the two methods can reconstruct the behaviors of the target program. The reconstruction performance is calculated as one minus the normalized Levenshtein Distance between the state sequences from the execution trace of the target program and from the execution trace of the synthesized program.

Table 3: **Learning to synthesize out-of-distributional programs.** HPRL demonstrates superior performance compared to LEAPS. The gap between the two methods grows more significant when the length of the target program increases.

Method	Program Reconstruction Performance			
	Len 25	Len 50	Len 75	Len 100
LEAPS	0.59 (0.14)	0.31 (0.10)	0.20 (0.05)	0.13 (0.08)
HPRL	**0.60** (0.03)	**0.34** (0.03)	**0.29** (0.03)	**0.26** (0.02)
Improvement	1.69%	9.68%	45.0%	100.0%

The result is presented in Table 3. HPRL consistently outperforms LEAPS with varying target program lengths, and the gap between the two methods grows more significant when the target program becomes longer. We also observe that the reconstruction score of LEAPS drops significantly as the length of target programs exceeds 40, which is the maximum program length of the program datasets. This suggests that HPRL can synthesize out-of-distributional programs. Note that the performance of HPRL can be further improved when setting the horizon of the meta-policy $|H|$ to a larger number. Yet, for this experiment, we fix it to 5 to better analyze our method. More details about the implementation and the evaluation metrics can be found in Section G.

4.4.2 DIMENSIONALITY OF PROGRAM EMBEDDING SPACE

Learning a higher-dimensional program embedding space can lead to better optimization in the program reconstruction loss and the latent behavior reconstruction loss. Yet, learning a meta-policy in a higher-dimensional action space can be unstable and inefficient. To investigate this trade-off and verify our contribution of employing the compression encoder f_ω and compression decoder g_ψ, we experiment with various dimensions of program embedding space and report the result in Table 4.

Table 4: **Dimensionality of the Program Embedding Space.** The 64-dimensional program embedding space demonstrates the best task performance with satisfactory reconstruction results.

dim(z)	Reconstruction		Task Performance	
	Program	Execution	CLEANHOUSE	SEEDER
16	81.70%	63.21%	0.47 (0.06)	0.21 (0.02)
32	94.46%	86.00%	0.84 (0.27)	0.35 (0.16)
64	97.81%	95.58%	1.00 (0.00)	0.58 (0.10)
128	99.12%	98.76%	1.00 (0.00)	0.57 (0.03)
256	99.65%	99.11%	1.00 (0.00)	0.54 (0.11)

The reconstruction accuracy measures if learned encoders and decoders can perfectly reconstruct an input program or its execution trace. The program embedding space with different dimensionalities are also evaluated in terms of task performance in CLEANHOUSE and SEEDER since they are considered more difficult. The result indicates that a 64-dimensional program embedding space achieves satisfactory reconstruction accuracy and performs the best on the tasks. Therefore, we take this dimension as the default setting for our proposed method.

5 CONCLUSION

We propose a hierarchical programmatic reinforcement learning framework, dubbed HRPL, which re-formulates solving a reinforcement learning task as synthesizing a task-solving program that can be executed to interact with the environment and maximize the return. Specifically, we first learn a program embedding space that continuously parameterizes a diverse set of programs sampled from a program dataset generated based on our proposed program generation procedure. Then, we train a meta-policy, whose action space is the learned program embedding space, to produce a series of programs (*i.e.*, predict a series of actions) to yield a composed task-solving program. Experimental results in the Karel domain on two problem sets demonstrate that our proposed framework consistently outperforms baselines by large margins. Ablation studies justify our design choices, including the reinforcement learning algorithms used to learn the meta-policy, and the dimensionality of the program embedding space.

Bibliography

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Association for the Advancement of Artificial Intelligence*, 2017.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 2003.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Neural Information Processing Systems*, 2018.

Rudy R Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, 2005.

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, 2019.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, 2017.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and Cooperation in Neural Nets: Proceedings of the US-Japan Joint Seminar*, 1982.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation*, 2017.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 2018.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2016.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, 2021.

Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 2021.

Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017.

Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning*, 2021.

Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward Hu, and Joseph J. Lim. Composing complex skills by learning transition policies. In *International Conference on Learning Representations*, 2019.

Youngwoon Lee, Andrew Szot, Shao-Hua Sun, and Joseph J. Lim. Generalizable imitation learning from observation via inferring goal proximity. In *Neural Information Processing Systems*, 2021.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 2022.

Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. Synthesizing environment-aware activities via activity sketches. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *International Conference on Language Resources and Evaluation*, 2018.

Zachary C Lipton. The mythos of model interpretability. In *ICML Workshop on Human Interpretability in Machine Learning*, 2016.

Guan-Ting Liu, Guan-Yu Lin, and Pu-Jen Cheng. Improving generalization with cross-state behavior matching in deep reinforcement learning. In *Autonomous Agents and Multiagent Systems*, 2022.

Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. Learning to describe scenes with programs. In *International Conference on Learning Representations*, 2019.

Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

Erika Puiutta and Eric M. S. P. Veith. Explainable reinforcement learning: A survey. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar R. Weippl (eds.), *Machine Learning and Knowledge Extraction - International Cross-Domain Conference, CD-MAKE*, 2020.

Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 1997.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Neural Information Processing Systems*, 2018.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *Association for the Advancement of Artificial Intelligence*, 2020.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, 2018.

Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 1999.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *International Conference on Learning Representations*, 2019.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems*, 2021.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 2018.

Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Neural Information Processing Systems*, 2019.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 2019.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

Elly Winner and Manuela Veloso. Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*, 2003.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Peter R Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, et al. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*, 2022.

Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

APPENDIX

A RELATED WORK

Program Synthesis. Program synthesis methods concern automatically synthesize programs that can transform some inputs to desired outputs. Encouraging results have been achieved in a variety of domains, including string transformation (Devlin et al., 2017; Hong et al., 2021), array/tensor transformation (Balog et al., 2017; Ellis et al., 2020), computer commands (Lin et al., 2018; Chen et al., 2021; Li et al., 2022), graphics and 3D shape programs (Wu et al., 2017; Liu et al., 2019; Tian et al., 2019), and describing behaviors of agents (Bunel et al., 2018; Sun et al., 2018; Shin et al., 2018; Chen et al., 2019; Liao et al., 2019; Silver et al., 2020). Most existing program synthesis methods consider task specifications such as input/output pairs, demonstrations, or natural language descriptions. In contrast, we aim to synthesize programs as policies that can be executed to induce behaviors which maximize rewards defined by reinforcement learning tasks.

Programmatic Reinforcement Learning. Programmatic reinforcement learning methods (Choi & Langley, 2005; Winner & Veloso, 2003; Sun et al., 2020) explore various programmatic and more structured representations of policies, including decision trees (Bastani et al., 2018), state machines Inala et al. (2020), symbolic expressions (Landajuela et al., 2021), and programs drawn from a domain-specific language Silver et al. (2020); Verma et al. (2018; 2019). Our work builds upon Trivedi et al. (2021), whose goal is to produce program policies from rewards. We aim to address the fundamental limitations of this work by learning to compose programs to yield more expressive programs.

Hierarchical Reinforcement Learning. Hierarchical reinforcement learning (HRL) Sutton et al. (1999); Barto & Mahadevan (2003); Vezhnevets et al. (2017); Bacon et al. (2017) aims to learn to operate on different levels of temporal abstraction, allowing for learning or exploring more efficiently in sparse-reward environments. In this work, instead of operating on pre-defined or learned temporal abstraction, we are interested in learning with a level of abstraction defined by a learned program embedding space to hierarchically compose programs. One can view a learned program embedding space as continuously parameterized options or low-level policies.

B LEARNING FROM EPISODIC REWARD

We design our framework to synthesize a sequence of programs, allowing for accurately rewarding correct programs and penalizing wrong programs (*i.e.*, better credit assignment) with dense rewards. In this section, we design experiments to investigate the effectiveness of this design. To this end, instead of receiving a reward for executing each program (*i.e.*, *dense*) in the environment, we modify CLEANHOUSE and SEEDER so that they only return cumulative rewards after all $|H|$ programs have been executed (*i.e.*, *episodic*). The learning performance is shown in Figure 4, demonstrating that learning from *dense* rewards yields better sample efficiency compared to learning from *episodic* rewards. This performance gain is made possible by the hierarchical design of HPRL, which can better deal with credit assignments. In contrast, LEAPS Trivedi et al. (2021) is fundamentally limited to learning from episodic rewards.

C METHOD DETAILS

The details of each method are described in this section.

C.1 DRL

The DRL method implements a deep neural network trained on the PPO algorithm for $2M$ time steps to learn the policy that takes the raw states (grids) from the Karel environment as input and predicts the next action. The raw state is a binary tensor representing the state of each grid.

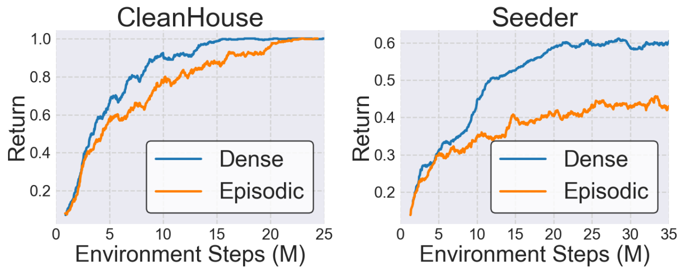


Figure 4: **Learning from Episodic Reward.** We compare learning from dense and episodic rewards in CLEANHOUSE and SEEDER. Learning from dense rewards achieves better sample efficiency in both tasks, which is made possible by the hierarchical design of our proposed framework.

C.2 DRL-ABS

DRL-abs is a deep neural network utilizing a recurrent policy and trained on the PPO algorithm due to the better performance compared with SAC. It is also trained for $2M$ time steps. The input is the abstract state of the Karel environment instead of the Karel raw states (grids). The abstract states are represented by [`frontIsClear() == True`, `leftIsClear()==False`, `rightIsClear()==True`, `markerPresent()==False`, `noMarkersPresent()==True`], which is a binary vector that describes the current state.

C.3 VIPER

VIPER is a programmatic RL framework proposed by Bastani et al. (2018) that uses a decision tree to imitate the behavior of a given neural network teacher policy. Bastani et al. (2018) takes the best DRL policy networks as its teacher policy. Since VIPER is not capable of synthesizing looping behaviors, it can be used to testify other approaches that employ a program embedding space to synthesize more complex programs.

C.4 LEAPS

LEAPS is a programmatic RL framework proposed by Trivedi et al. (2021). The training framework includes two stages. First, it trains a model with an encoder-decoder architecture to learn a continuous program embedding space. The second stage utilizes the Cross-Entropy Method Rubinstein (1997), searching over the leaned program embedding space to optimize the program policy for each task.

C.5 LEAPS-OURS

LEAPS-ours uses the same framework as LEAPS but is trained on our proposed dataset when learning a program embedding space.

C.6 HPRL

The overall framework of HPRL consists of two parts: The pre-trained decoder as mentioned in Section 3.1 and the meta-policy described in Section 3.3. The decoder is constructed with a one-layer unidirectional GRU with hidden size and input size set to 256. We further compress the latent program space consisting of a fully-connected linear neural network with an input dimension of 256 and an output dimension of $[16, 32, 64, 128]$. Please note that VAE with 256 dimensions does not include the fully connected linear neural network. The meta policy neural network consists of the CNN neural network as a state feature extractor and a fully-connected linear layer for the action and value branch. The CNN neural network includes two convolutional layers. The filter size of the first convolutional layer is 32 with 4 channel. The filter size of the second convolutional layer is 32 with 2 channel. The output of the state embedding is flattened to a vector of the same size as the output action vector.

The pseudocode of HPRL is shown at Algorithm 1.

Algorithm 1 Hierarchical Programmatic Reinforcement Learning

Input: Program Dataset $D_{program}$, VAE Training Epoch N_{epoch}, Meta-Policy Training Step T_{meta}, Program Synthesis Step $|H|$
Output: Task Solving Program $\mathcal{P}$

```
 1: Initialize the program encoder $q_\phi$ and decoder $p_\theta$, compression encoder $f_\omega$ and decoder $g_\psi$,
    neural execution policy $\pi$.
 2: for epoch in range(1, $N_{epoch}$) do
 3:     for program $\rho$ in $D_{program}$ do
 4:         $z = f_\omega(q_\phi(\rho))$
 5:         $\hat{\rho} = p_\theta(g_\psi(z))$
 6:         compute $\mathcal{L}_{total} = \mathcal{L}^P_{\theta,\phi,\omega,\phi}(\rho) + \lambda\mathcal{L}^L_\pi(\rho, \pi)$
 7:         fit $\phi, \omega, \psi, \theta, \pi$ to minimize $\mathcal{L}_{total}$
 8:     end for
 9: end for
10: Initialize a meta-policy $\pi_{meta}$
11: Load and fix $p_\theta, g_\psi$ for Meta-Policy Training
12: for Training Episode in range(1, $T_{meta}/|H|$) do
13:     Receive initial state $s^1$ from the Karel environment
14:     for i in range(1, $|H|$) do
15:         $z_i = \pi_{meta}(s^i)$
16:         $\rho_i = p_\theta(g_\psi(z_i))$
17:         Interact with the environment by EXEC($\rho_i$)
18:         Receive $[s^i_1, ..., s^i_T]$ and $[r^i_1, ..., r^i_{T_i}]$
19:         $r^{i+1} = \sum_{t=1}^{T^i} r^i_t$
20:         $s^{i+1} = [s^i_1, s^i_2, ..., s^i_{T_i}][-1]$
21:     end for
22:     Calculate $\mathcal{J}_{\pi_{meta}}$ based on the collected $(s^i, z^i, r^{i+1}, s^{i+1})$
23:     fit $\pi_{meta}$ to maximize $\mathcal{J}_{\pi_{meta}}$
24: end for
```

D Problem Set Details

D.1 Karel Problem Set Details

The KAREL problem set introduced in Trivedi et al. (2021) consists of six different tasks: STAIRCLIMBER, FOURCORNER, TOPOFF, MAZE, CLEANHOUSE and HARVESTER. The performance of the policy networks is measured by averaging the rewards of 10 random environment initial configurations. All experiments are tested on 8×8 grid except for CLEANHOUSE. Figure 5 visualizes the ideal end states and one of their random initial configurations of all tasks.

D.2 StairClimber

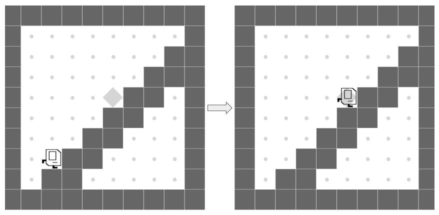
In this task, the agent is asked to move along the stair to reach the marked grid. The initial location of the agent and the marker are randomized near the stair with a marker on the higher end. The reward is defined as 1 if the agent reaches the marked grid, and 0 otherwise.

D.3 FourCorner

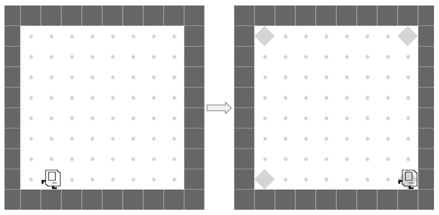
The goal of the agent is to place a marker on each corner to earn the reward. Once any marker is placed in the wrong location, the reward is 0. The reward is the number of corrected placed markers multiplied by 0.25. The initial position of the agent is at the last row of the environment facing east.

D.4 TopOff

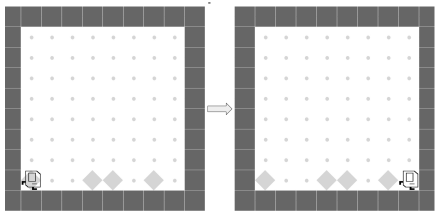
The agent is asked to place markers on marked grids and reach the destination on the rightest grid of the bottom row in this task. The reward is defined as the consecutive correct states of the last



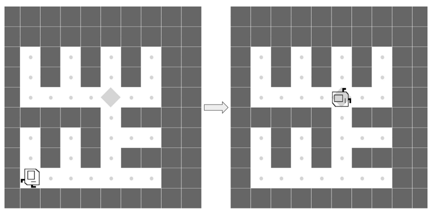
(a) STAIRCLIMBER



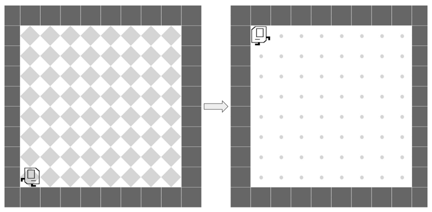
(b) FOURCORNER



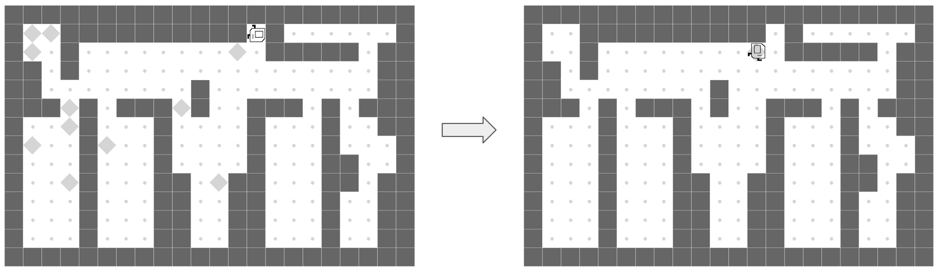
(c) TOPOFF



(d) MAZE



(e) HARVESTER



(f) CLEANHOUSE

Figure 5: Illustrations of the initial and desired final state of each task in the KAREL Problem set introduced in by Trivedi et al. (2021). Note that these illustrations are from (Trivedi et al., 2021). The position of markers, walls, and agent's position are randomly set according to the configurations of each tasks. More details are provided in Section D.1.

rows until the agent puts a marker on an empty location or does not place a marker on a marked grid. If the agent ends up on the rightest grid of the last row, a bonus reward is given. The agent is always initiated on the leftist grid of the bottom row, while the locations of markers in the last row are randomized.

D.5 MAZE

In this task, the agent has to navigate to reach the marked destination. The locations of markers and the agent, as well as the configuration of the maze, are randomized. The reward is 1 if the agent successfully reaches the marked grid or otherwise 0.

D.6 CLEANHOUSE

There is some garbage (markers) around the apartment, so the agent is asked to clean them up. The agent will receive more rewards for collecting more markers on the grid. A grid is of size 14×22, which represents an apartment. The location of the agent is fixed, while the marker locations are randomized. The reward is defined as the number of markers picked to divide the total number of markers in the initial Karel state.

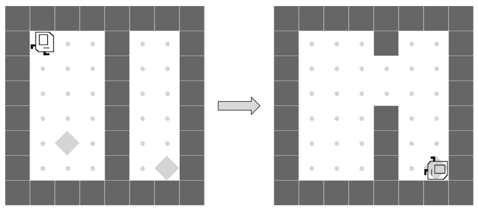
D.7 HARVESTER

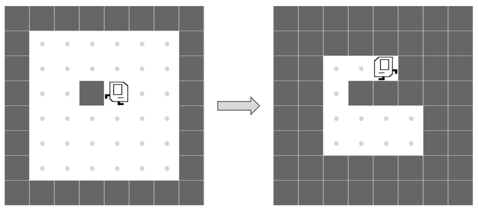
The goal is to collect more markers on the grid, with markers appearing in all grids in the initial Karel environment. The reward is defined as the number of collected markers divided by the total markers in the initial state.

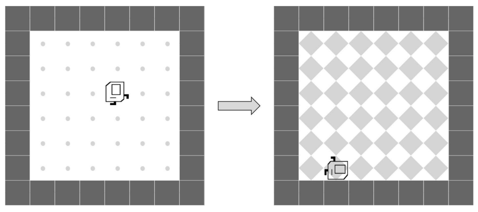
D.8 KAREL-HARD PROBLEM SET DETAILS

Since all the tasks in the original Karel benchmark are well-solved by our method, we proposed a newly designed Karel-Hard benchmark to further evaluate the capability of HPRL. We define the state transition functions and reward functions for DOORKEY, ONESTROKE, SEEDER, and SNAKE based on Karel states. Each task includes more constraints and more complex structures, *e.g*. two-phase structure for DOORKEY, the restriction of no revisiting for ONESTROKE.

The performance of the policy networks is measured by averaging the rewards of 10 random environment initial configurations. The range of cumulative reward in all KAREL-HARD tasks is $[0.0, 1.0]$. Figure 6 visualize the ideal end states and one of their random initial configurations of all tasks.


(a) DOORKEY


(b) ONESTROKE


(c) SEEDER

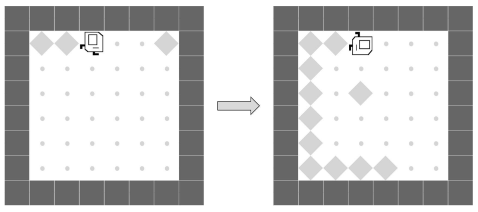

(d) SNAKE

Figure 6: Illustrations of the initial and final state of each task in the proposed KAREL-HARD Problem Set. The position of markers, walls, and agent's position are randomly set according to the configurations of each tasks. More details are provided in Section D.8.

D.9 DOORKEY

An 8×8 grid is split into two areas: a 6×3 left chamber and a 6×2 right chamber. The two chambers are unconnected in the beginning. The agent has to pick up the marker in the left chamber to unlock the door, and then get into the right chamber to place the marker on the top of the target(marker). The initial location of the agent, the key(marker) in the left room and the target(marker) in the right room are randomly initialized. The reward is defined as 0.5 for picking up the key and the other 0.5 for placing the marker on the marked grid.

D.10 ONESTROKE

The goal is to make the agent traverse all grids without revisiting. The visited grids will become a wall and the episode will terminate if the agent hits the wall. The reward is defined as the number of grids visited divided by the total empty grids in the initial Karel environment. The initial location of the agent is randomized.

D.11 SEEDER

The goal is to put markers on each grid in the Karel environment. The episode will end if makers are repeatedly placed. The reward is defined as the number of markers placed divided by the total empty grids in the initial Karel environment. The initial location of the agent is randomized.

D.12 SNAKE

In this task, the agent acts like the head of the snake, and the goal is to eat (*i.e.*, pass through) as much food(markers) as possible without hitting its body. There is always exactly one marker existing in the environment until 20 markers are eaten. Once the agent passes the marker, the snake body length will increase by 1, and one new marker will appear on the other position of the environment. The reward is defined as the number of markers eaten divided by 20. The locations where the markers will appear are fixed, while the initial agent location is randomized.

E HYPERPARAMETERS AND SETTINGS

E.1 LEAPS

Following the setting of LEAPSTrivedi et al. (2021), we experiment with sets of hyperparameters when searching the program embedding space to optimize the reward for both LEAPS and LEAPS-ours. The settings are described in Table 5 and Table 6. S, σ, # Elites, Exp Decay and D_I represent population size, standard deviation, exponential σ decay and initial distribution, respectively.

Karel-Hard tasks

Table 5: LEAPS experiment settings on KAREL-HARD tasks.

LEAPS	S	σ	# Elites	Exp Decay	D_I
DOORKEY	32	0.25	0.1	False	$N(0, 0.1I_d)$
ONESTROKE	64	0.5	0.05	True	$N(1, 0)$
SEEDER	32	0.25	0.1	False	$N(0, 0.1I_d)$
SNAKE	32	0.25	0.2	False	$N(0, I_d)$

Reconstruction tasks

Table 6: LEAPS experiment settings on Program Reconstruction tasks.

LEAPS	S	σ	# Elites	Exp Decay	D_I
Len 25	32	0.5	0.05	True	$N(0, I_d)$
Len 50	32	0.5	0.2	True	$N(0, 0.1I_d)$
Len 75	64	0.5	0.05	True	$N(0, 0.1I_d)$
Len 100	64	0.5	0.1	True	$N(0, 0.1I_d)$

E.2 LEAPS-OURS

Karel tasks

Table 7: LEAPS-ours experiment settings on KAREL tasks.

LEAPS-ours	S	σ	# Elites	Exp Decay	D_I
STAIRCLIMBER	32	0.5	0.05	True	$N(0, 0.1I_d)$
FOURCORNERS	32	0.5	0.1	True	$N(1, 0)$
TOPOFF	64	0.25	0.05	Trie	$N(0, 0.1I_d)$
MAZE	64	0.1	0.2	False	$N(1, 0)$
CLEANHOUSE	64	0.1	0.05	False	$N(1, 0)$
HARVESTER	64	0.5	0.05	True	$N(1, 0)$

Karel-Hard tasks

Table 8: LEAPS-ours experiment settings on KAREL-HARD tasks.

LEAPS-ours	S	σ	# Elites	Exp Decay	D_I
DOORKEY	64	0.5	0.2	True	$N(1, 0)$
ONESTROKE	64	0.5	0.05	False	$N(0, I_d)$
SEEDER	32	0.25	0.1	False	$N(1, 0)$
SNAKE	32	0.25	0.05	False	$N(0, 0.1, I_d)$

E.3 HPRL

Pretraining VAE

- Latent embedding size: 64
- GRU Hidden Layer Size: 256
- # GRU layer for encoder/decoder: 1
- Batch Size: 256
- Nonlinearity: $Tanh()$
- Optimizer: Adam
- Learning Rate: 0.001
- Latent Loss Coefficient (β): 0.1

RL training on Meta Policies

The Hyperparameters for HPRL-PPO and HPRL-SAC training are reported in Table 9. For each task, we test on 3 different random seeds and take the average to measure the performance.

F THE KAREL PROGRAM DATASETS GENERATION

The Karel program dataset used in this work includes 1 million program sequences, with 85% as the training dataset and 15% as the evaluation dataset. In addition to sequences of program tokens, the KAREL program dataset also includes execution demonstrations (*e.g.* state transition and action sequence) of each program in the dataset, which can be used for the *latent behavior reconstruction loss* described in Section 3.1.

To further improve the data quality, we added some heuristic rules while selecting data to filter out the programs with repetitive or offsetting behavior. The unwanted programs that we drop while collecting data are mainly determined by the following rules:

- Contradictory Primitive Actions: `turnLeft` followed by `turnRight`, `pickMarker` followed by `putMarker`, or vice versa.
- Meaningless Programs: `end_state == start_state` after program execution
- Repetitive behaviors: a program that has the longest common subsequence of tokens longer than 9

We further analyze the distribution of the generated program sequences based on the control flow (e.g., `IF, IFELSE`) and loop command (e.g., `WHILE, REPEAT`). The statistical probabilities of programs containing control flow or loop commands are listed in Table 10. Results show that more than 40% of the programs in the collected program sequences contain at least one of the control of loop commands, ensuring the diversity of the generated programs.

Table 9: Hyperparameters of HPRL-PPO and HPRL-SAC Training

Training Settngs	SAC	PPO
Max # Subprogram	5	5
Max Subprogram Length	40	40
Batch Size	1024	256
Specific Parameters	Init. Temperatur: 0.0002 Actor Update Frequency: 200 Critic Target Update Frequency: 200 Num Seed Steps: 20000 Reply Buffer Size: 5M Training Steps: 25M Alpha Learning Rate: 0.0001 Actor Learning Rate: 0.0001 Critic Learning Rate: 0.00001 β: [0.9, 0.999] Critic τ: 0.005 Number of parallel actors: 16 Discount factor: 0.99 Q-critic Hidden Dimension: 16	Learning Rate: 0.00005 Entropy Coefficient: 0.1 Rollout Size: 12800 Eps: 0.00001 α: 0.99 γ: 0.99 Use GAE: True GAE lambda: 0.95 Value Loss Coefficient: 0.5 Clip Param: 0.2 max grad. norm.: 0.5 Update Epoch: 3 clip param.: 0.2 Training Steps: 25M

Table 10: The statistical distribution of programs containing each token in our generated dataset.

	IFELSE	IF	WHILE	REPEAR
Our Dataset	41%	47%	54%	22%

G More on Learning to Synthesize Out-of-distributional Programs

Measure the Performance. To measure the performance of programs synthesized by different methods, we first collect and execute each target program, yielding a target state sequence $\tau_{target} = [s_1, s_2, \ldots, s_{T_{target}}]$. Then, we reset the Karel environment to the initial state s_1. For our proposed framework, we synthesize a sequence of programs with the following procedure and optimize a program reconstruction reward to match the target program. As described in Section 3.3, at each macro training time step $n, 1 \leq n \leq |H|$, we collect the state sequence $\tau_{\mathcal{P}} = [s_1^{\mathcal{P}}, s_2^{\mathcal{P}}, \ldots, s_{T_{\mathcal{P}}}^{\mathcal{P}}]$ from the executing of task-solving program $\mathcal{P} = \langle \rho_i | i = 1, .., n \rangle$, and calculate the program reconstruction reward $r^n = 1 - \mathcal{D}(\tau_{target}, \tau_{\mathcal{P}})$ where $\mathcal{D}$ is the normalized Levenshtein distance. For executing the programs synthesized by LEAPS and LEAPS-ours, we simply start executing programs after resetting the Karel environment to the initial state s_1 and calculating the return. A Python implementation that calculates the program reconstruction performance is as follows.

```
import numpy as np

def _compare_demos(demo1, demo1_len, demo2, demo2_len):
    if demo2_len == 0: return 0
    if demo1_len == 1 and demo2_len == 1: return 1
    distances = np.zeros((demo1_len + 1, demo2_len + 1))
    for t1 in range(demo1_len + 1):
        distances[t1][0] = t1
    for t2 in range(demo2_len + 1):
        distances[0][t2] = t2
    a = 0
    b = 0
    c = 0
    for t1 in range(1, demo1_len + 1):
        for t2 in range(1, demo2_len + 1):
            if np.array_equal(demo1[t1-1], demo2[t2-1]):
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                a = distances[t1][t2 - 1]
                b = distances[t1 - 1][t2]
                c = distances[t1 - 1][t2 - 1]
                if (a <= b and a <= c):
                    distances[t1][t2] = a + 1
                elif (b <= a and b <= c):
                    distances[t1][t2] = b + 1
                else:
                    distances[t1][t2] = c + 1
    return 1.0 - ((distances[demo1_len][demo2_len]) / (max(demo1_len,
        demo2_len)-1))
```

H SYNTHESIZED PROGRAMS

In this section, we provide qualitative results (*i.e.*, synthesized programs) of our proposed framework (HPRL-PPO), LEAPS, and LEAPS-ours. The programs synthesized for the tasks in the KAREL problem set are shown in Figure 7 (STAIRCLIMBER, TOPOFF) Figure 8 (CLEANHOUSE), Figure 9 (FOURCORNER, MAZE), and Figure 10 (HARVESTER). The programs synthesized for the tasks in the KAREL-HARD problem set are shown in Figure 11 (DOORKEY), Figure 12(ONESTROKE), and Figure 13 (SEEDER and SNAKE).

Karel Programs

STAIRCLIMBER

LEAPS

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        move
        w)
    WHILE c( rightIsClear c) w
          (
        turnLeft
        w)
    m)
```

LEAPS-ours

```
DEF run m(
    turnRight
    turnRight
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        move
        w)
    m)
```

HPRL-PPO

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        move
        turnRight
        move
        w)
    m)
```

TOPOFF

LEAPS

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        move
        w)
    putMarker
    move
    WHILE c( not c(
          markersPresent c) c)
           w(
        move
        w)
    putMarker
    move
    WHILE c( not c(
          markersPresent c) c)
           w(
        move
        w)
    putMarker
    move
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    m)
```

LEAPS-ours

```
DEF run m(
    WHILE c( not c(
          rightIsClear c) c) w
          (
        WHILE c( not c(
              markersPresent c
              ) c) w(
            move
            w)
        putMarker
        move
        w)
    WHILE c( not c(
          rightIsClear c) c) w
          (
        pickMarker
        w)
    m)
```

HPRL-PPO

```
DEF run m(
    move
    move
    REPEAT R=5 r(
        move
        WHILE c(
              noMarkersPresent
              c) w(
            move
            w)
        putMarker
        r)
    m)
```

Figure 7: **Example programs on Karel tasks: STAIRCLIMBER and TOPOFF.** The programs with best rewards out of all random seeds are shown.

CLEANHOUSE

LEAPS

```
DEF run m(
    WHILE c( noMarkersPresent
         c) w(
        turnRight
        move
        move
        turnLeft
        turnRight
        pickMarker
        w)
    turnLeft
    turnRight
    m)
```

LEAPS-ours

```
DEF run m(
    move
    WHILE c( noMarkersPresent
         c) w(
        turnRight
        move
        WHILE c( frontIsClear
             c) w(
            move
            pickMarker
            w)
        w)
    m)
```

HPRL-PPO

```
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            REPEAT R=4 r(
                turnRight
                move
                pickMarker
                move
                r)
            r)
        r)
    m)
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            REPEAT R=4 r(
                REPEAT R=4 r(
                    turnRight
                    move
                    pickMarker
                    move
                    r)
                r)
            r)
        r)
    m)
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            REPEAT R=4 r(
                pickMarker
                move
                turnRight
                move
                r)
            r)
        r)
    m)
```

Figure 8: **Example programs on Karel tasks: CLEANHOUSE.** The programs with best rewards out of all random seeds are shown.

FOURCORNER

LEAPS

```
DEF run m(
    turnRight
    move
    turnRight
    turnRight
    turnRight
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnRight
    putMarker
    m)
```

LEAPS-ours

```
DEF run m(
    REPEAT R=5 r(
        WHILE c( frontIsClear
             c) w(
            move
            w)
        IFELSE c( not c(
             rightIsClear c)
             c) i(
            turnLeft
            putMarker
            i)
        ELSE e(
            putMarker
            e)
        r)
    m)
```

HPRL-PPO

```
DEF run m(
    move
    move
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnLeft
    putMarker
    m)
DEF run m(
    move
    move
    WHILE c( frontIsClear c) w
          (
        move
        w)
    putMarker
    WHILE c( frontIsClear c) w
          (
        move
        w)
    turnLeft
    m)
DEF run m(
    move
    move
    WHILE c( frontIsClear c) w
          (
        move
        w)
    putMarker
    turnLeft
    putMarker
    WHILE c( frontIsClear c) w
          (
        move
        w)
    putMarker
    putMarker
    m)
```

MAZE

LEAPS

```
DEF run m(
    IF c( frontIsClear c) i(
        turnLeft
        i)
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        move
        w)
    m)
```

LEAPS-ours

```
DEF run m(
    turnRight
    turnRight
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        move
        w)
    m)
```

HPRL-PPO

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        move
        turnRight
        w)
    move
    m)
```

Figure 9: **Example programs on Karel tasks: FOURCORNER and MAZE.** The programs with best rewards out of all random seeds are shown.

HARVESTER

LEAPS

```
DEF run m(
    turnLeft
    turnLeft
    pickMarker
    move
    pickMarker
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    m)
```

LEAPS-ours

```
DEF run m(
    WHILE c( leftIsClear c) w(
        REPEAT R=4 r(
            pickMarker
            move
            r)
        turnLeft
        pickMarker
        move
        turnLeft
        pickMarker
        move
        w)
        turnLeft
        pickMarker
        turnLeft
    m)
```

HPRL-PPO

```
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            pickMarker
            turnRight
            move
            pickMarker
            turnRight
            move
            pickMarker
            move
            pickMarker
            move
            r)
        turnRight
        pickMarker
        move
        pickMarker
        move
        pickMarker
        move
        r)
    m)
```

Figure 10: **Example programs on Karel tasks: HARVESTER.** The programs with best rewards out of all random seeds are shown.

Karel-Hard Programs

DOORKEY

LEAPS

```
DEF run m(
    move
    turnRight
    putMarker
    pickMarker
    move
    WHILE c( leftIsClear c) w(
        pickMarker
        move
        w)
    m)
```

LEAPS-ours

```
DEF run m(
    WHILE c( rightIsClear c) w
          (
        turnRight
        pickMarker
        turnLeft
        pickMarker
        pickMarker
        pickMarker
        pickMarker
        move
        turnLeft
        move
        w)
    m)
```

HPRL-PPO

```
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
             turnRight move
             pickMarker move
             pickMarker move
             r)
        pickMarker move r)
    m)
DEF run m(
    REPEAT R=5 r(
        turnRight move
        REPEAT R=5 r( move r)
        move pickMarker move
        r)
    m)
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            turnRight move
            REPEAT R=3 r(
                move
                     pickMarker

                move
                     pickMarker
                      r)
            r)
        r)
    m)
DEF run m(
    REPEAT R=4 r(
        REPEAT R=4 r(
            turnRight move
            pickMarker move
            pickMarker
            REPEAT R=2 r(
                pickMarker
                     move
                pickMarker
                     pickMarker
                      r)
            r)
        r)
    m)
DEF run m(
    REPEAT R=4 r(
        turnRight
        REPEAT R=4 r(
            turnRight move
                   move
            pickMarker move
            r)
        move pickMarker move
        r)
    move pickMarker
    m)
```

Figure 11: **Example programs on Karel-Hard tasks: DOORKEY.** The programs with best rewards out of all random seeds are shown.

ONESTROKE

LEAPS

```
DEF run m(
    REPEAT R=9 r(
        turnRight
        turnRight
        WHILE c( frontIsClear
              c) w(
            move
            w)
        turnRight
        WHILE c( frontIsClear
              c) w(
            move
            w)
        r)
    turnRight
    m)
```

LEAPS-ours

```
DEF run m(
    turnRight
    WHILE c( frontIsClear c) w
          (
        WHILE c( frontIsClear
              c) w(
            WHILE c(
                  frontIsClear
                   c) w(
                WHILE c(
                      frontIsClear
                       c) w(
                    move
                    w)
                turnRight
                w)
            turnRight
            w)
        turnRight
        w)
    turnRight
    m)
```

HPRL-PPO

```
DEF run m(
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    m)
DEF run m(
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    m)
DEF run m(
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    m)
DEF run m(
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    WHILE c( frontIsClear c) w
          ( move w) turnRight
    m)
```

Figure 12: **Example programs on Karel-Hard tasks: ONESTROKE.** The programs with best rewards out of all random seeds are shown.

SEEDER

LEAPS

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        turnRight
        putMarker
        move
        move
        w)
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    m)
```

LEAPS-ours

```
DEF run m(
    WHILE c( noMarkersPresent
          c) w(
        putMarker
        move
        turnRight
        move
        w)
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    turnRight
    m)
```

HPRL-PPO

```
DEF run m(
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    turnRight move
    m)
DEF run m(
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    turnRight move
    putMarker move
    m)
DEF run m(
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    turnRight move
    putMarker move
    turnRight move
    m)
DEF run m(
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    turnRight move
    putMarker move
    turnRight move
DEF run m(
    putMarker move
    putMarker move
    putMarker move
    putMarker move
    turnRight move
    putMarker move
    turnRight move
    m)
```

SNAKE

LEAPS

```
DEF run m(
    turnRight
    turnLeft
    pickMarker
    move
    move
    move
    WHILE c( rightIsClear c) w
          (
        turnLeft
        move
        move
        w)
    turnLeft
    turnLeft
    turnLeft
    turnLeft
    m)
```

LEAPS-ours

```
DEF run m(
    move
    turnRight
    pickMarker
    pickMarker
    WHILE c( rightIsClear c) w
          (
        turnLeft
        move
        move
        w)
    turnRight
    move
    move
    move
    m)
```

HPRL-PPO

```
DEF run m(
    move
    WHILE c( noMarkersPresent
          c) w(
        move
        move
        turnLeft
        w)
    move
    turnLeft
    m)
DEF run m(
    move
    WHILE c( noMarkersPresent
          c) w(
        move
        move
        turnLeft
        w)
    m)
DEF run m(
    move
    WHILE c( noMarkersPresent
          c) w(
        move
        move
        turnLeft
        w)
    move
    turnLeft
    m)
```

Figure 13: **Example programs on Karel-Hard tasks: SEEDER and SNAKE.** The programs with the best reward out of all random seeds are shown.